

APPENDIX 3

```

/**
 * IPS rule processing functions
 * @file rules.h
 * @author jmccaskey
 */

#ifndef RULES__H
#define RULES__H

/**
 * Function to process all the rules related to a single monitor.
 * The function will select all the rules for the monitor and then evaluate them in turn.
 * It will insert any necessary error_log entries and update escalations as it goes.
 */
int process_rules(monitor *mon, long double gauge, MYSQL *mysql_connection);

/**
 * Function to adjust timestamp to users timezone from UTC timestamp.
 * The function should return 0 for success 1 for failure.
 */
int timezone_adjust(struct tm **tm_struct, int timezone_offset, int daylight);

/**
 * Function to test whether a rule (as represented by a MYSQL_ROW object) is active for the specified
 * tm structure (which should be the polling time in the users timezone). Function returns 1 if the rule
 * passed the check, 0 otherwise.
 */
int test_date(struct tm *tm_struct, MYSQL_ROW row);

/**
 * Function to format a value to appropriate unit increment. It is assumed that the value coming in
 * is already in the base unit (ie bps, or packets, not kbps or (thou)packets). adjusted_unit will
 * be dynamically allocated and the caller must free it at a later point.
 */
long double format_value(char *adjusted_unit, long double old_value, char *old_unit);

/**
 * Function to test whether a static rule is violated.
 */
void test_static_threshold(unsigned int rule_id, unsigned int rule_server_id, monitor *mon,
                           char *operator, long double value, long double poll_value);

/**
 * Function to test whether a variable rule is violated.
 */
void test_variable_threshold(int rule_id, int rule_server_id, monitor *mon,
                             char *operator, long double value, long double poll_value, char *value_type,
                             int time_value, char *time_unit, MYSQL *mysql_connection);

/**
 * Function to test whether a cumulative rule is violated.
 */
void test_cumulative_threshold(int rule_id, int rule_server_id, monitor *mon,
                               char *operator, long double value, long double poll_value, int percent,
                               int time_value, char *time_unit, MYSQL *mysql_connection);

```

```
/**
 * Function to test whether a stddev rule is violated.
 */
void test_stddev_threshold(int rule_id, int rule_server_id, monitor *mon,
                           char *operator, long double value, long double poll_value, int time_value,
                           char *time_unit, MYSQL *mysql_connection);
```

```
#include "rules.c"
```

```
#endif
```

```

/**
 * IPS rule processing functions
 * @file rules.c
 * @author jmccaskey
 */

/**
 * Function to process all the rules related to a single monitor.
 * The function will select all the rules for the monitor and then evaluate them in turn.
 * It will insert any necessary error_log entries and update escalations as it goes.
 */
int process_rules(monitor *mon, long double gauge, MYSQL *mysql_connection) {
    //select all the rules for the monitor
    MYSQL_RES *result;
    MYSQL_ROW row;
    int n;
    char *sql_query;
    assert(sql_query=malloc(1000));
    n=sprintf(sql_query, 1000, "SELECT rule.rule_id, rule.rule_server_id, threshold_type,
timeframe_start, timeframe_stop, "
"timeframe_all, monday, tuesday, wednesday, thursday, friday, saturday, sunday, primary_email,
primary_pager, primary_escalation_delay, "
"secondary_email, secondary_pager, secondary_escalation_delay, tertiary_email, tertiary_pager,
tertiary_escalation_delay, current_escalation, "
"offset, daylight FROM rule, rule_monitor, user, account, timezone "
"WHERE rule.rule_id = rule_monitor.rule_id "
"AND rule_monitor.monitor_id = %d "
"AND rule_monitor.monitor_server_id = %d "
"AND rule.user_id = user.user_id "
"AND rule.user_server_id = user.user_server_id "
"AND account.account_id = user.account_id "
"AND account.account_server_id = user.account_server_id "
"AND user.timezone_id = timezone.timezone_id "
"AND user.timezone_server_id = timezone.timezone_server_id "
"AND rule.active='on' ", mon->monitor_id, mon->monitor_server_id);

    if(mysql_real_query(mysql_connection, sql_query, n)!=0) {
        fprintf(stderr, "Failed while attempting to select rules: Error: %s\n",
mysql_error(mysql_connection));
        free(sql_query);
        return(1);
    }
    free(sql_query);

    //store results from query
    result=mysql_store_result(mysql_connection);

    //loop through all the rows checking each rule as we go
    while(row=mysql_fetch_row(result)) {
        /**
         * check if the rule currently applies (ie turned on for current day/time)
         */
        //copy the timestamp to a new variable and adjust it to the users local timezone
        struct tm *tm_struct;
        if(timezone_adjust(&tm_struct, atoi(row[23]), atoi(row[24]))) {
            //the timezone adjustment failed, don't evaluate this rule...

```

```

        continue;
    }

    if(test_date(tm_struct, row)) {
        MYSQL_RES *results_extra;
        MYSQL_ROW row_extra;
        //the rule is active for this polling period, perform checking
#ifdef DEBUG
        flockfile(stdout);
        fprintf(stdout, "Evaluating Rule: %s, %s\n", row[0], row[1]);
        funlockfile(stdout);
#endif

        //check which type of rule it is and select any additional type specific parameters
        if(strcmp(row[2], "static")==0) {
            assert(sql_query = malloc(1000));
            n=snprintf(sql_query, 1000, "SELECT operator, value FROM rule_static "
                "WHERE rule_id=%s AND rule_server_id=%s LIMIT 1", row[0],
row[1]);

            if(mysql_real_query(mysql_connection, sql_query, n)!=0) {
                fprintf(stderr, "Failed while attempting to select rule details: Error:
%s\n", mysql_error(mysql_connection));
                free(sql_query);
                continue; //continue on to the next rule maybe it will work
            }
            free(sql_query);

            //store results from query
            results_extra=mysql_store_result(mysql_connection);

            if(row_extra=mysql_fetch_row(results_extra)) {
                //call actual static evaluation here
                test_static_threshold(atoi(row[0]), atoi(row[1]), mon,
row_extra[0],
                                atof(row_extra[1]), gauge);
            }
            mysql_free_result(results_extra);
        } else if(strcmp(row[2], "variable")==0) {
            assert(sql_query = malloc(1000));
            n=snprintf(sql_query, 1000, "SELECT operator, value, time_value,
time_unit, value_type FROM rule_variable "
                "WHERE rule_id=%s AND rule_server_id=%s LIMIT 1", row[0],
row[1]);

            if(mysql_real_query(mysql_connection, sql_query, n)!=0) {
                fprintf(stderr, "Failed while attempting to select rule details: Error: %s\n",
mysql_error(mysql_connection));
                free(sql_query);
                continue; //continue on to the next rule maybe it will work
            }
            free(sql_query);

            //store results from query
            results_extra=mysql_store_result(mysql_connection);

            if(row_extra=mysql_fetch_row(results_extra)) {
                //call actual variable evaluation here
                test_variable_threshold(atoi(row[0]), atoi(row[1]), mon,

```

```

row_extra[0], atof(row_extra[1]), gauge, row_extra[4],
                                                                    atoi(row_extra[2]), row_extra[3],
mysql_connection);
    }
    mysql_free_result(results_extra);
} else if(strcmp(row[2], "cumulative")==0) {
    assert(sql_query = malloc(1000));
    n=snprintf(sql_query, 1000, "SELECT operator, value, time_value,
time_unit, percent FROM rule_cumulative "
                                "WHERE rule_id=%s AND rule_server_id=%s LIMIT 1", row[0],
row[1]);
    if(mysql_real_query(mysql_connection, sql_query, n)!=0) {
        fprintf(stderr, "Failed while attempting to select rule details: Error: %s\n",
mysql_error(mysql_connection));
        free(sql_query);
        continue; //continue on to the next rule maybe it will work
    }
    free(sql_query);

    //store results from query
    results_extra=mysql_store_result(mysql_connection);

    if(row_extra=mysql_fetch_row(results_extra)) {
        //call actual cumulative evaluation here
        test_cumulative_threshold(atoi(row[0]), atoi(row[1]), mon,
row_extra[0], atof(row_extra[1]), gauge, atoi(row_extra[4]),
        atoi(row_extra[2]), row_extra[3], mysql_connection);
    }
    mysql_free_result(results_extra);
} else if(strcmp(row[2], "stddev")==0) {
    assert(sql_query = malloc(1000));
    n=snprintf(sql_query, 1000, "SELECT operator, value, time_value,
time_unit FROM rule_stddev "
                                "WHERE rule_id=%s AND rule_server_id=%s LIMIT 1", row[0],
row[1]);
    if(mysql_real_query(mysql_connection, sql_query, n)!=0) {
        fprintf(stderr, "Failed while attempting to select rule details: Error: %s\n",
mysql_error(mysql_connection));
        free(sql_query);
        continue; //continue on to the next rule maybe it will work
    }
    free(sql_query);

    //store results from query
    results_extra=mysql_store_result(mysql_connection);

    if(row_extra=mysql_fetch_row(results_extra)) {
        //call actual stddev evaluation here
        test_stddev_threshold(atoi(row[0]), atoi(row[1]), mon,
row_extra[0], atof(row_extra[1]), gauge,
                                                                    atoi(row_extra[2]), row_extra[3], mysql_connection);
    }
    mysql_free_result(results_extra);
}

//(should actually go in individual functions) insert error_log entries if needed...

```

```

    }
}
mysql_free_result(result);
return(0);
}

/**
 * Function to adjust timestamp to users timezone from UTC timestamp.
 * The function should return 0 for success 1 for failure.
 */
int timezone_adjust(struct tm **tm_struct, int timezone_offset, int daylight) {
    if(daylight) {
        timezone_offset += 1;
    }
    if(timezone_offset!=0) {
        //set user time in seconds since 1970 to match polling period start time...
        time_t utime = rawtime;
        //adjust value based off timezone info... +/- n hours
        utime += timezone_offset*3600;
        //get tm structure from utime value
        *tm_struct = localtime(&utime);
    }
    return(0);
}

/**
 * Function to test whether a rule (as represented by a MYSQL_ROW object) is active for the specified
 * tm structure (which should be the polling time in the users timezone). Function returns 1 if the rule
 * passed the check, 0 otherwise.
 */
int test_date(struct tm *tm_struct, MYSQL_ROW row) {
    int dotw_passed = 0;
#ifdef DEBUG
    flockfile(stdout);
    fprintf(stdout, "UserTime: Day of Week: %d, Hour: %d\n", tm_struct->tm_wday, tm_struct->tm_hour);
    funlockfile(stdout);
#endif
    //check what day it is, and whether the bit for that day is set in the rule we are checking
    if(tm_struct->tm_wday==0) {
        if(strcmp(row[12], "on")==0) {
            dotw_passed = 1;
        }
    } else if(tm_struct->tm_wday==1) {
        if(strcmp(row[6], "on")==0) {
            dotw_passed = 1;
        }
    } else if(tm_struct->tm_wday==2) {
        if(strcmp(row[7], "on")==0) {
            dotw_passed = 1;
        }
    } else if(tm_struct->tm_wday==3) {
        if(strcmp(row[8], "on")==0) {
            dotw_passed = 1;
        }
    } else if(tm_struct->tm_wday==4) {

```

```

        if(strcmp(row[9], "on")==0) {
            dotw_passed = 1;
        }
        } else if(tm_struct->tm_wday==5) {
            if(strcmp(row[10], "on")==0) {
                dotw_passed = 1;
            }
        } else if(tm_struct->tm_wday==6) {
            if(strcmp(row[11], "on")==0) {
                dotw_passed = 1;
            }
        }
    }
    if(dotw_passed==1) {
        //check if the rule is set for timeframe_all, if so check has passed
        if(strcmp(row[5], "on")==0)
            return 1;
        if(tm_struct->tm_hour >= atoi(row[3]) && tm_struct->tm_hour < atoi(row[4]))
            return 1;
        else
            return 0;
    } else {
        return 0;
    }
}

/**
 * Function to convert time_value from rules into seconds based off time_unit.
 */
unsigned long int time_to_seconds(int time_value, char *time_unit) {
    unsigned long int seconds = 0;
    if(strcmp(time_unit, "hours")==0)
        seconds = time_value * 3600;
    else if(strcmp(time_unit, "days")==0)
        seconds = time_value * 86400;
    else
        seconds = time_value;
    return seconds;
}

/**
 * Function to format a value to appropriate unit increment. It is assumed that the value coming in
 * is already in the base unit (ie bps, or packets, not kbps or (thou)packets). adjusted_unit will
 * be dynamically allocated and the caller must free it at a later point.
 */
long double format_value(char *adjusted_unit, long double old_value, char *old_unit)
{
    if(strcmp(old_unit, "available")==0) {
        sprintf(adjusted_unit, "available");
        return old_value;
    }
    long double adjusted_value = 0;
    if(strcmp(old_unit, "bits")==0 || strcmp(old_unit, "bps")==0) {
        if(old_value > (long double)1023
            && old_value < (long double)1024*(long double)1024) {
            adjusted_value = old_value/1024;
            sprintf(adjusted_unit, "k%s", old_unit);
        }
    }
}

```



```

    } else if(old_value > ((long double)1024*(long double)1024)-1
    && old_value < (long double)1024*(long double)1024*(long double)1024) {
        adjusted_value = old_value/((long double)1024*(long double)1024);
        sprintf(adjusted_unit, "m%s", old_unit);
    } else if(old_value > ((long double)1024*(long double)1024*(long double)1024)-1
    && old_value < (long double)1024*(long double)1024*(long double)1024*(long
double)1024) {
        adjusted_value = old_value/((long double)1024*(long double)1024*(long
double)1024);
        sprintf(adjusted_unit, "g%s", old_unit);
    } else if(old_value > ((long double)1024*(long double)1024*(long double)1024*(long
double)1024)-1) {
        adjusted_value = old_value/((long double)1024*(long double)1024*(long
double)1024*(long double)1024);
        sprintf(adjusted_unit, "t%s", old_unit);
    } else {
        adjusted_value = old_value;
        sprintf(adjusted_unit, "%s", old_unit);
    }
} else if(strcmp(old_unit, "B")==0) {
    if(old_value > (long double)1023
    && old_value < (long double)1024*(long double)1024) {
        adjusted_value = old_value/1024;
        sprintf(adjusted_unit, "K%s", old_unit);
    } else if(old_value > ((long double)1024*(long double)1024)-1
    && old_value < (long double)1024*(long double)1024*(long double)1024) {
        adjusted_value = old_value/((long double)1024*(long double)1024);
        sprintf(adjusted_unit, "M%s", old_unit);
    } else if(old_value > ((long double)1024*(long double)1024*(long double)1024)-1
    && old_value < (long double)1024*(long double)1024*(long double)1024*(long double)1024) {
        adjusted_value = old_value/((long double)1024*(long double)1024*(long double)1024);
        sprintf(adjusted_unit, "G%s", old_unit);
    } else if(old_value > ((long double)1024*(long double)1024*(long double)1024*(long
double)1024)-1) {
        adjusted_value = old_value/((long double)1024*(long double)1024*(long double)1024*(long
double)1024);
        sprintf(adjusted_unit, "T%s", old_unit);
    } else {
        adjusted_value = old_value;
        sprintf(adjusted_unit, "%s", old_unit);
    }
} else {
    if(old_value > (long double)999 && old_value < (long double)1000000) {
        adjusted_value = old_value/(long double)1000;
        sprintf(adjusted_unit, "(thou) %s", old_unit);
    } else if(old_value > (long double)999999 && old_value < (long double)1000000000) {
        adjusted_value = old_value/(long double)1000000;
        sprintf(adjusted_unit, "(mil) %s", old_unit);
    } else if(old_value > (long double)999999999 && old_value < (long
double)1000000000*(long double)10000) {
        adjusted_value = old_value/((long double)1000000*(long double)1000);
        sprintf(adjusted_unit, "(bil) %s", old_unit);
    } else if(old_value > ((long double)1000000000*(long double)1000)-1) {
        adjusted_value = old_value/(long double)1000000000*(long double)1000;
        sprintf(adjusted_unit, "(tril) %s", old_unit);
    } else {

```

```

        adjusted_value = old_value;
        sprintf(adjusted_unit, "%s", old_unit);
    }
}
return adjusted_value;
}

/**
 * Function to test whether a static rule is violated.
 */
void test_static_threshold(unsigned int rule_id, unsigned int rule_server_id, monitor *mon,
                          char *operator, long double value, long double poll_value)
{
#ifdef DEBUG
    flockfile(stdout);
    fprintf(stdout, "Static Rule Value: %Lf Polled Value: %Lf Divisor: %f\n", value, poll_value, mon-
>divisor);
    funlockfile(stdout);
#endif

    //setup the error node
    error_node *errnode;
    assert(errnode = malloc(sizeof(*errnode)));
    errnode->rule_id = rule_id;
    errnode->rule_server_id = rule_server_id;
    errnode->monitor_id = mon->monitor_id;
    errnode->monitor_server_id = mon->monitor_server_id;

    //make sure we aren't going to be dividing by zero if someone stupidly put a 0 in for a metric in the
db!
    if(mon->divisor == 0)
        mon->divisor = 1;
    poll_value /= mon->divisor;

    char value_unit_adjusted[strlen(mon->unit)+10];
    long double value_adjusted;
    value_adjusted = format_value(value_unit_adjusted, value, mon->unit);

    char poll_value_unit_adjusted[strlen(mon->unit)+10];
    long double poll_value_adjusted;
    poll_value_adjusted = format_value(poll_value_unit_adjusted, poll_value, mon->unit);

    if(strcmp(operator, "falls below")==0 && poll_value < value) {
        errnode->failed = 1;
        snprintf(errnode->message, sizeof(errnode->message),
                  "Falls below static threshold of %1.2Lf %s (Value: %1.2Lf %s)",
                  value_adjusted, value_unit_adjusted, poll_value_adjusted,
poll_value_unit_adjusted);
    } else if(strcmp(operator, "exceeds")==0 && poll_value > value) {
        errnode->failed = 1;
        snprintf(errnode->message, sizeof(errnode->message),
                  "Exceeds static threshold of %1.2Lf %s (Value: %1.2Lf %s)",
                  value_adjusted, value_unit_adjusted, poll_value_adjusted,
poll_value_unit_adjusted);
    } else if(strcmp(operator, "equals")==0 && poll_value == value) {
        errnode->failed = 1;

```

```

        snprintf(errnode->message, sizeof(errnode->message),
            "Equals static threshold of %1.2Lf %s (Value: %1.2Lf %s)",
                value_adjusted, value_unit_adjusted, poll_value_adjusted,
poll_value_unit_adjusted);
    } else if(strcmp(operator, "unavailable")==0 && poll_value < 1) {
        errnode->failed = 1;
        snprintf(errnode->message, sizeof(errnode->message), "Fails static availability threshold:
Device was unavailable.");
    } else {
        //rule is not violated
        errnode->failed = 0;
        if(strcmp(operator, "exceeds")==0)
            snprintf(errnode->message, sizeof(errnode->message),
                "Does not exceed static threshold of %1.2Lf %s (Value: %1.2Lf %s)",
                    value_adjusted, value_unit_adjusted, poll_value_adjusted,
poll_value_unit_adjusted);
        else if(strcmp(operator, "falls below")==0)
            snprintf(errnode->message, sizeof(errnode->message),
                "Does not fall below static threshold of %1.2Lf %s (Value: %1.2Lf %s)",
                    value_adjusted, value_unit_adjusted, poll_value_adjusted,
poll_value_unit_adjusted);
        else if(strcmp(operator, "equals")==0)
            snprintf(errnode->message, sizeof(errnode->message),
                "Does not equal static threshold of %1.2Lf %s (Value: %1.2Lf %s)",
                    value_adjusted, value_unit_adjusted, poll_value_adjusted,
poll_value_unit_adjusted);
        else if(strcmp(operator, "unavailable")==0)
            snprintf(errnode->message, sizeof(errnode->message), "Passes static availability
threshold: Device was available.");
        else
            snprintf(errnode->message, sizeof(errnode->message), "");
    }

    //push error_node
    pthread_mutex_lock(&error_work_queue.mutex);
    queue_put(&error_work_queue.c_queue, (queue_node *)errnode);
    pthread_mutex_unlock(&error_work_queue.mutex);
    pthread_cond_broadcast(&error_work_queue.cond);

    return;
}

/**
 * Function to test whether a variable rule is violated.
 */
void test_variable_threshold(int rule_id, int rule_server_id, monitor *mon,
                            char *operator, long double value, long double poll_value, char
*value_type,
                            int time_value, char *time_unit, MYSQL *mysql_connection) {
    //make sure we don't divide by zero because someones put a bad value in the metrics table!
    if(mon->divisor==0)
        mon->divisor = 1;

#ifdef DEBUG
    flockfile(stdout);
    fprintf(stdout, "Variable Rule Value: %Lf Polled Value: %Lf Divisor: %f Time Value: %d Time Unit:
%s\n",

```

```

        value, poll_value, mon->divisor, time_value, time_unit);
    funlockfile(stdout);
#endif
    //setup the error node
    error_node *errnode;
    assert(errnode = malloc(sizeof(*errnode)));
    errnode->rule_id = rule_id;
    errnode->rule_server_id = rule_server_id;
    errnode->monitor_id = mon->monitor_id;
    errnode->monitor_server_id = mon->monitor_server_id;
    errnode->failed = 0;

    //evaluate the rule and generate the errnode->failed and errnode->message values
    char *sql_query;
    int n;
    MYSQL_RES *result;
    MYSQL_ROW row;

    unsigned long int seconds;
    long double average;
    seconds = time_to_seconds(time_value, time_unit);

    assert(sql_query = malloc(800));
    n=sprintf(sql_query, 800, "SELECT AVG(gauge) AS gauge FROM event_log WHERE "
        "monitor_id=%d AND monitor_server_id=%d AND timestamp >=
DATE_SUB(NOW(), INTERVAL %d SECOND) "
        "AND timestamp <= NOW() GROUP BY monitor_id", mon->monitor_id,
mon->monitor_server_id, seconds);
    if(mysql_real_query(mysql_connection, sql_query, n)!=0) {
        flockfile(stderr);
        fprintf(stderr, "Failed while attempting to select AVG for variable rule... aborting rule evaluation:
Error: %s\n", mysql_error(mysql_connection));
        funlockfile(stderr);
        free(errnode);
        free(sql_query);
        return;
    }
    free(sql_query);

    //store results from last query into result
    result=mysql_store_result(mysql_connection);
    row=mysql_fetch_row(result);
    if(row==NULL) {
        flockfile(stderr);
        fprintf(stderr, "Couldn't fetch average for variable rule, aborting rule evaluation...\n");
        funlockfile(stderr);
        free(errnode);
        mysql_free_result(result);
        return;
    }

    average = atof(row[0]);
    mysql_free_result(result);

    if(strcmp(value_type, "unit")==0) {
#ifdef DEBUG

```

```

        flockfile(stdout);
        fprintf(stdout, "Testing Var: Avg: %Lf Polled: %Lf\n", average, poll_value);
        funlockfile(stdout);
#endif

        long double test_value;
        //figure out the delta from the average over the period
        test_value = (poll_value - average) / mon->divisor;

        char value_unit_adjusted[strlen(mon->unit)+10];
        long double value_adjusted;
        value_adjusted = format_value(value_unit_adjusted, value, mon->unit);

        //char test_value_unit_adjusted[strlen(mon->unit)+10];
        //long double test_value_adjusted;
        //test_value_adjusted = format_value(test_value_unit_adjusted, test_value, mon->unit);

        if(strcmp(operator, "increases")==0 && test_value > value) {
            errnode->failed = 1;
            snprintf(errnode->message, sizeof(errnode->message), "Increased above
variable threshold of "
                        "%1.2Lf %s in %d %s", value_adjusted, value_unit_adjusted, time_value,
time_unit);
        } else if(strcmp(operator, "decreases")==0 && test_value < (-1*value)) {
            errnode->failed = 1;
            snprintf(errnode->message, sizeof(errnode->message), "Decreased below
variable threshold of "
                        "%1.2Lf %s in %d %s", value_adjusted, value_unit_adjusted, time_value,
time_unit);
        } else {
            errnode->failed = 0;
            if(strcmp(operator, "increases")==0) {
                snprintf(errnode->message, sizeof(errnode->message), "Passes variable
threshold: Did not increase "
                        "%1.2Lf %s in %d %s", value_adjusted, value_unit_adjusted,
time_value, time_unit);
            } else if(strcmp(operator, "decreases")==0) {
                snprintf(errnode->message, sizeof(errnode->message), "Passes variable
threshold: Did not decrease "
                        "%1.2Lf %s in %d %s", value_adjusted, value_unit_adjusted,
time_value, time_unit);
            }
        }
    } else if(strcmp(value_type, "percent")==0) {
        double percent;
        //figure out the percentage change from the average over the period
        if(average==0) {
            percent = 0;
        } else {
            percent = (poll_value - average) / average * 100;
        }
#ifdef DEBUG
        flockfile(stdout);
        fprintf(stdout, "Testing Var: Avg: %Lf PollVal: %Lf PctChange: %f ThresholdPct: %Lf\n",
average, poll_value, percent, value);
        funlockfile(stdout);
#endif
    }
}

```

```

        if(strcmp(operator, "increases")==0 && percent >= value) {
            errnode->failed = 1;
            snprintf(errnode->message, sizeof(errnode->message), "Increased by more than
the variable threshold of %1.2Lf%% in %d %s",
                    value, time_value, time_unit);
        } else if(strcmp(operator, "decreases")==0 && (-1*percent) >= value) {
            errnode->failed = 1;
            snprintf(errnode->message, sizeof(errnode->message), "Decreased by more
than the variable threshold of %1.2Lf%% in %d %s",
                    value, time_value, time_unit);
        } else {
            errnode->failed = 0;
            if(strcmp(operator, "increases")==0) {
                snprintf(errnode->message, sizeof(errnode->message), "Passes variable
threshold: Did not increase "
                        "by more than %1.2Lf%% in %d %s", value, time_value,
time_unit);
            } else if(strcmp(operator, "decreases")==0) {
                snprintf(errnode->message, sizeof(errnode->message), "Passes variable
threshold: Did not decrease "
                        "by more than %1.2Lf%% in %d %s", value, time_value,
time_unit);
            }
        }
    }
}

```

```

    //push error_node
    pthread_mutex_lock(&error_work_queue.mutex);
    queue_put(&error_work_queue.c_queue, (queue_node *)errnode);
    pthread_mutex_unlock(&error_work_queue.mutex);
    pthread_cond_broadcast(&error_work_queue.cond);

    return;
}

/**
 * Function to test whether a cumulative rule is violated.
 */
void test_cumulative_threshold(int rule_id, int rule_server_id, monitor *mon,
                              char *operator, long double value, long double poll_value, int percent,
                              int time_value, char *time_unit, MYSQL *mysql_connection) {
    //make sure we don't divide by zero because someones put a bad value in the metrics table!
    if(mon->divisor==0)
        mon->divisor = 1;
#ifdef DEBUG
    flockfile(stdout);
    fprintf(stdout, "Cumulative Rule Value: %Lf Polled Value: %Lf Percent: %d Divisor: %f Time Value:
%d Time Unit: %s\n",
            value, poll_value, percent, mon->divisor, time_value, time_unit);
    funlockfile(stdout);
#endif
    //setup the error node
    error_node *errnode;
    assert(errnode = malloc(sizeof(*errnode)));
    errnode->rule_id = rule_id;
    errnode->rule_server_id = rule_server_id;
}

```

```

errnode->monitor_id = mon->monitor_id;
errnode->monitor_server_id = mon->monitor_server_id;
errnode->failed = 0;

//evaluate the rule and generate the errnode->failed and errnode->message values
char *sql_query;
int n;
MYSQL_RES *result;
MYSQL_ROW row;

unsigned long int seconds;
    int violation_count;
    float percent_violated;
    char symbol;
seconds = time_to_seconds(time_value, time_unit);

    //convert availability rules into normal ones...
    if(strcmp(operator, "available")==0) {
        value = 0;
    } else if(strcmp(operator, "unavailable")==0) {
        value = 1;
    }

    if(strcmp(operator, "exceeds")==0 || strcmp(operator, "available")==0) {
        symbol = '>';
    } else if(strcmp(operator, "falls below")==0 || strcmp(operator, "unavailable")==0) {
        symbol = '<';
    } else {
        symbol = '=';
    }

assert(sql_query = malloc(800));
n=snprintf(sql_query, 800, "SELECT COUNT(*) AS violation_count FROM event_log WHERE "
    "monitor_id=%d AND monitor_server_id=%d AND timestamp >= DATE_SUB(NOW(),
INTERVAL %d SECOND) "
    "AND timestamp <= NOW() AND gauge %c %Lf", mon->monitor_id, mon-
>monitor_server_id, seconds, symbol, value);
if(mysql_real_query(mysql_connection, sql_query, n)!=0) {
    flockfile(stderr);
    fprintf(stderr, "Failed while attempting to select violation count for cumulative rule ...
aborting rule evaluation: Error: %s\n", mysql_error(mysql_connection));
    funlockfile(stderr);
    free(errnode);
    free(sql_query);
    return;
}
free(sql_query);

//store results from last query into result
result=mysql_store_result(mysql_connection);
row=mysql_fetch_row(result);
if(row==NULL) {
    flockfile(stderr);
    fprintf(stderr, "Couldn't fetch violation count for cumulative rule, aborting rule evaluation...\n");
    funlockfile(stderr);
    free(errnode);

```

```

        mysql_free_result(result);
        return;
    }

    violation_count = atoi(row[0]);
    mysql_free_result(result);

    percent_violated = (float)violation_count / (seconds / 300) * 100;

    //lockfile(stdout);
    //fprintf(stdout, "vio count: %d percent vio: %f\n", violation_count, percent_violated);
    //unlockfile(stdout);

    char value_unit_adjusted[strlen(mon->unit)+10];
    long double value_adjusted;
    value_adjusted = format_value(value_unit_adjusted, value, mon->unit);

    if(strcmp(operator, "falls below")==0 && percent_violated >= percent) {
        errnode->failed = 1;
        snprintf(errnode->message, sizeof(errnode->message), "Fails cumulative threshold: Fell
below %1.2Lf %s %1.2f "
                "percent of the time over last %d %s", value_adjusted, value_unit_adjusted,
percent_violated, time_value, time_unit);
    } else if(strcmp(operator, "exceeds")==0 && percent_violated >= percent) {
        errnode->failed = 1;
        snprintf(errnode->message, sizeof(errnode->message), "Fails cumulative threshold:
Exceeded %1.2Lf %s %1.2f "
                "percent of the time over last %d %s", value_adjusted, value_unit_adjusted,
percent_violated, time_value, time_unit);
    } else if(strcmp(operator, "falls bellow")==0) {
        errnode->failed = 0;
        snprintf(errnode->message, sizeof(errnode->message), "Passes cumulative threshold:
Did not fall below %1.2Lf %s %d.00 "
                "percent of the time over last %d %s", value_adjusted, value_unit_adjusted,
percent, time_value, time_unit);
    } else if(strcmp(operator, "exceeds")==0) {
        errnode->failed = 0;
        snprintf(errnode->message, sizeof(errnode->message), "Passes cumulative threshold:
Did not exceed %1.2Lf %s %d.00 "
                "percent of the time over last %d %s", value_adjusted, value_unit_adjusted,
percent, time_value, time_unit);
    } else if(strcmp(operator, "available")==0 && percent_violated >= percent) {
        errnode->failed = 1;
        snprintf(errnode->message, sizeof(errnode->message), "Fails cumulative availability
threshold: Available %1.2f "
                "percent of the time over last %d %s", percent_violated, time_value, time_unit);
    } else if(strcmp(operator, "unavailable")==0 && percent_violated >= percent) {
        errnode->failed = 1;
        snprintf(errnode->message, sizeof(errnode->message), "Fails cumulative availability threshold:
Unavailable %1.2f "
                "percent of the time over last %d %s", percent_violated, time_value, time_unit);
    } else if(strcmp(operator, "available")==0) {
        errnode->failed = 0;
        snprintf(errnode->message, sizeof(errnode->message), "Passes cumulative availability
threshold: Available %1.2f "
                "percent of the time over last %d %s", percent_violated, time_value, time_unit);
    }

```



```

    } else if(strcmp(operator, "unavailable")==0) {
        errnode->failed = 0;
        snprintf(errnode->message, sizeof(errnode->message), "Passes cumulative availability
threshold: Unavailable %1.2f "
                "percent of the time over last %d %s", percent_violated, time_value, time_unit);
    }

    //push error_node
    pthread_mutex_lock(&error_work_queue.mutex);
    queue_put(&error_work_queue.c_queue, (queue_node *)errnode);
    pthread_mutex_unlock(&error_work_queue.mutex);
    pthread_cond_broadcast(&error_work_queue.cond);

    return;
}
/**
 * Function to test whether a stddev rule is violated.
 */
void test_stddev_threshold(int rule_id, int rule_server_id, monitor *mon,
                          char *operator, long double value, long double poll_value, int time_value,
                          char *time_unit, MYSQL *mysql_connection) {
    //make sure we don't divide by zero because someones put a bad value in the metrics table!
    if(mon->divisor==0)
        mon->divisor = 1;
#ifdef DEBUG
    flockfile(stdout);
    fprintf(stdout, "StdDev Rule Value: %Lf Polled Value: %Lf Divisor: %f Time Value: %d Time Unit:
%s\n",
            value, poll_value, mon->divisor, time_value, time_unit);
    funlockfile(stdout);
#endif
    //setup the error node
    error_node *errnode;
    assert(errnode = malloc(sizeof(*errnode)));
    errnode->rule_id = rule_id;
    errnode->rule_server_id = rule_server_id;
    errnode->monitor_id = mon->monitor_id;
    errnode->monitor_server_id = mon->monitor_server_id;
    errnode->failed = 0;

    //evaluate the rule and generate the errnode->failed and errnode->message values
    char *sql_query;
    int n;
    MYSQL_RES *result;
    MYSQL_ROW row;

    long double stddev_value;
    unsigned long int seconds;
    seconds = time_to_seconds(time_value, time_unit);

    assert(sql_query = malloc(800));
    n=snprintf(sql_query, 800, "SELECT AVG(gauge) AS mean, STDDEV(gauge) AS stddev FROM
event_log "
        "WHERE monitor_id=%d AND monitor_server_id=%d AND timestamp >=
DATE_SUB(NOW(), INTERVAL %d SECOND) "
        "AND timestamp <= NOW() GROUP BY monitor_id", mon->monitor_id,

```

```

mon->monitor_server_id, seconds);

if(mysql_real_query(mysql_connection, sql_query, n)!=0) {
    flockfile(stderr);
    fprintf(stderr, "Failed while attempting to select mean/stddev for stdev rule ... aborting rule
evaluation: Error: %s\n", mysql_error(mysql_connection));
    funlockfile(stderr);
    free(errno);
    free(sql_query);
    return;
}
free(sql_query);

//store results from last query into result
result=mysql_store_result(mysql_connection);
row=mysql_fetch_row(result);
if(row==NULL) {
    flockfile(stderr);
    fprintf(stderr, "Couldn't fetch violation count for cumulative rule, aborting rule evaluation...\n");
    funlockfile(stderr);
    free(errno);
    mysql_free_result(result);
    return;
}

if(strcmp(operator, "exceeds")==0)
    stddev_value = atof(row[0])+(value*atof(row[1]));
else
    stddev_value = atof(row[0])-(value*atof(row[1]));

mysql_free_result(result);

char stddev_value_unit_adjusted[strlen(mon->unit)+10];
long double stddev_value_adjusted;
stddev_value_adjusted = format_value(stddev_value_unit_adjusted, stddev_value/mon->divisor,
mon->unit);

char poll_value_unit_adjusted[strlen(mon->unit)+10];
long double poll_value_adjusted;
poll_value_adjusted = format_value(poll_value_unit_adjusted, poll_value/mon->divisor, mon->unit);

if(strcmp(operator, "exceeds")==0 && poll_value > stddev_value) {
    errno->failed = 1;
    snprintf(errno->message, sizeof(errno->message), "Fails standard deviation
threshold: %1.2Lf %s "
"exceeds mean + %1.2Lf stddev (%1.2Lf %s) as taken over %d %s", poll_value_adjusted,
poll_value_unit_adjusted,
value, stddev_value_adjusted, stddev_value_unit_adjusted, time_value,
time_unit);
} else if(strcmp(operator, "falls below")==0 && poll_value < stddev_value) {
    errno->failed = 1;
    snprintf(errno->message, sizeof(errno->message), "Fails standard deviation
threshold: %1.2Lf %s "
"falls below mean - %1.2Lf stddev (%1.2Lf %s) as taken over %d %s",
poll_value_adjusted, poll_value_unit_adjusted,

```

```

        value, stddev_value_adjusted, stddev_value_unit_adjusted, time_value,
time_unit);
    } else if(strcmp(operator, "exceeds")==0) {
        errnode->failed = 0;
        snprintf(errnode->message, sizeof(errnode->message), "Passes standard deviation
threshold: %1.2Lf %s "
        "does not exceed mean + %1.2Lf stddev (%1.2Lf %s) as taken over %d %s",
poll_value_adjusted, poll_value_unit_adjusted,
        value, stddev_value_adjusted, stddev_value_unit_adjusted, time_value,
time_unit);
    } else if(strcmp(operator, "falls below")==0) {
        errnode->failed = 0;
        snprintf(errnode->message, sizeof(errnode->message), "Passes standard deviation
threshold: %1.2Lf %s "
        "does not fall below mean - %1.2Lf stddev (%1.2Lf %s) as taken over %d %s",
poll_value_adjusted, poll_value_unit_adjusted,
        value, stddev_value_adjusted, stddev_value_unit_adjusted, time_value,
time_unit);
    }

    //push error_node
    pthread_mutex_lock(&error_work_queue.mutex);
    queue_put(&error_work_queue.c_queue, (queue_node *)errnode);
    pthread_mutex_unlock(&error_work_queue.mutex);
    pthread_cond_broadcast(&error_work_queue.cond);

    return;
}

```